NOTICE

- Deadlines
  - (5/13 11:59 PM) Programming assignment 3
  - (5/15 11:59 PM) Midterm Quiz 3
  - 5/18, 20: No classes; lecture recordings will be made available asynchronously

CS 344: Operating Systems I
Part IV – Synchronization I

Mon/Wed 12:00 – 1:50 PM

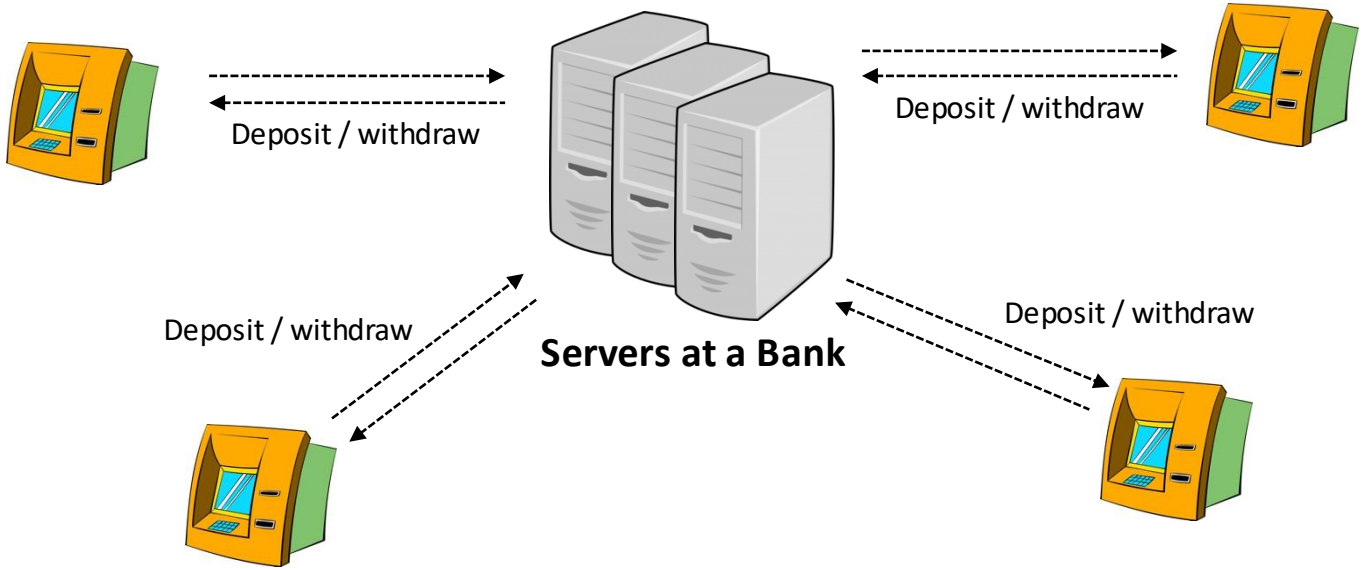
Sanghyun Hong
sanghyun.hong@oregonstate.edu

TOPICS FOR TODAY

- Part IV – Synchronization
  - Manage resources
    - Problem of sharing resources
    - Race condition
  - Provide abstraction
    - Atomic operation
    - Mutual exclusion (mutex)
  - Offer standard interface
    - Mutex C libraries
    - Critical section

SYNCHRONIZATION

- **ATM Bank Server**
  - The server(s) takes care of multiple deposit / withdrawal requests
  - Bank want to make sure all the transactions are correct



ATM BANK SERVER V0.1

- **One-at-a-time**
  - Receive a request
  - Process the request
  - Perform those actions *iteratively*

ATM BANK SERVER V0.1

- **One-at-a-time**
  - Receive a request
  - Process the request
  - Perform those actions *iteratively*

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         Deposit(accountId, amount);
      case OP_WITHDRAW:
         Withdraw(accountId, amount);
         … <here, you can define more ops…>
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId = -1;
   int amount = -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;        // code only reaches here if the server terminates
}
```

ATM BANK SERVER V0.1

- **One-at-a-time**
  - Receive a request
  - Process the request
  - Perform those actions *iteratively*
- **Problems**
  - 470k+ ATMs in the US (2018)
  - Requests need to wait
  - No scalability

```
void ProcessRequest(op, accountId, amount) {
  switch (op) {
    case OP_DEPOSIT:
      Deposit(accountId, amount);
    case OP_WITHDRAW:
      Withdraw(accountId, amount);
      … <here, you can define more ops…>
  }
}

void Deposit(accountId, amount) {
  account = GetAccount(accountId);
  account->balance += amount;
  StoreAccount(account);
}

int main(void) {
  int op = -1;
  int accountId = -1;
  int amount = -1;

  while (1) {
    ReceiveRequest(&op, &accountId, &amount);
    ProcessRequest(op, accountId, amount);
  }

  return 0;        // code only reaches here if the server terminates
}
```

ATM BANK SERVER V0.2

- **Event-driven**
  - Receive/process events
  - Store them to a queue (or a buffer)
  - Process them asynchronously

ATM BANK SERVER V0.2

- **Event-driven**
  - Receive/process events
  - Store them to a queue (or a buffer)
  - Process them asynchronously

```
struct Event {
   int eventType;
   int accountId;
   int amount;
   struct account* account;
};

void PullAccount(struct Event* event) {
   event->account = GetAccount(event->accountId);
}

void Deposit(struct Event* event) {
   event->account->balance += event->amount;
   event->amount = 0;
}

int main(void) {
   …

   while (1) {
      event = Wait4NextEvent();       // save new event in a Q; process
      switch (event->eventType) {
         case RequestReceived: PullAccount(&e); break;
         case DepositReady:    Deposit(&e);     break;
      }
   }
   return 0;          // code only reaches here if the server terminates
}
```

ATM BANK SERVER V0.2

- **Event-driven**
  - Receive/process events
  - Store them to a queue (or a buffer)
  - Process them asynchronously
- **Potential problem:**
  - Implementation complexity
  - Management complexity
    - How many event types?
    - How large a queue should be?
    - …

```
struct Event {
   int eventType;
   int accountId;
   int amount;
   struct account* account;
};

void PullAccount(struct Event* event) {
   event->account = GetAccount(event->accountId);
}

void Deposit(struct Event* event) {
   event->account->balance += event->amount;
   event->amount = 0;
}

int main(void) {
   …

   while (1) {
      event = Wait4NextEvent();       // save new event in a Q; process
      switch (event->eventType) {
         case RequestReceived: PullAccount(&e); break;
         case DepositReady:    Deposit(&e);     break;
      }
   }
   return 0;          // code only reaches here if the server terminates
}
```

ATM BANK SERVER V0.3

- **Multi-threaded**
  - Receive a request
  - Create a thread for processing it
  - Process multiple request concurrently

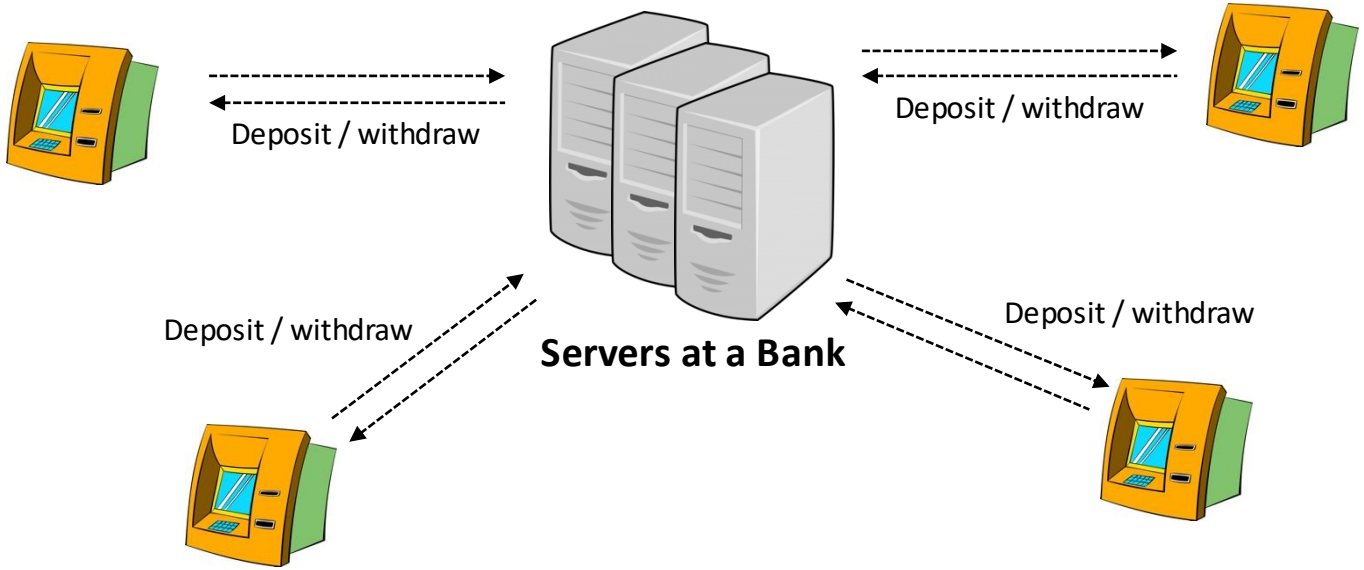
ATM BANK SERVER V0.3

- **Multi-threaded**
  - Receive a request
  - Create a thread for processing it
  - Process multiple request concurrently

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Deposit, info);
      case OP_WITHDRAW:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Withdraw, info);
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
}
```

ATM BANK SERVER V0.3

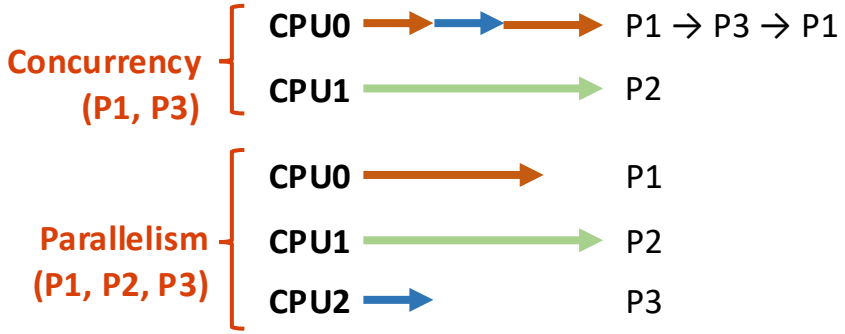
- **Multi-threaded ATM bank servers**
  - The servers take care of multiple requests
  - Multiple processes are processed concurrently



NOTE ON TERMINOLOGY

- **Concurrency vs. parallelism:**
  - Concurrency: handling multiple processes (or threads) at once
  - Parallelism: running multiple processes (or threads) *simultaneously*

- **Example:**
  - On the CPU0
    - P1 and P3 can execute *concurrently*
    - P1 and P3 is *not* running in parallel
  - On the CPU0 and CPU1
    - P1 and P2 runs in parallel



PROBLEM OF MULTI-THREADED BANK SERVER

- **OS controls the concurrency**
  - Can't control context switch
  - Can't control the execution order
  - Can't guarantee the balance

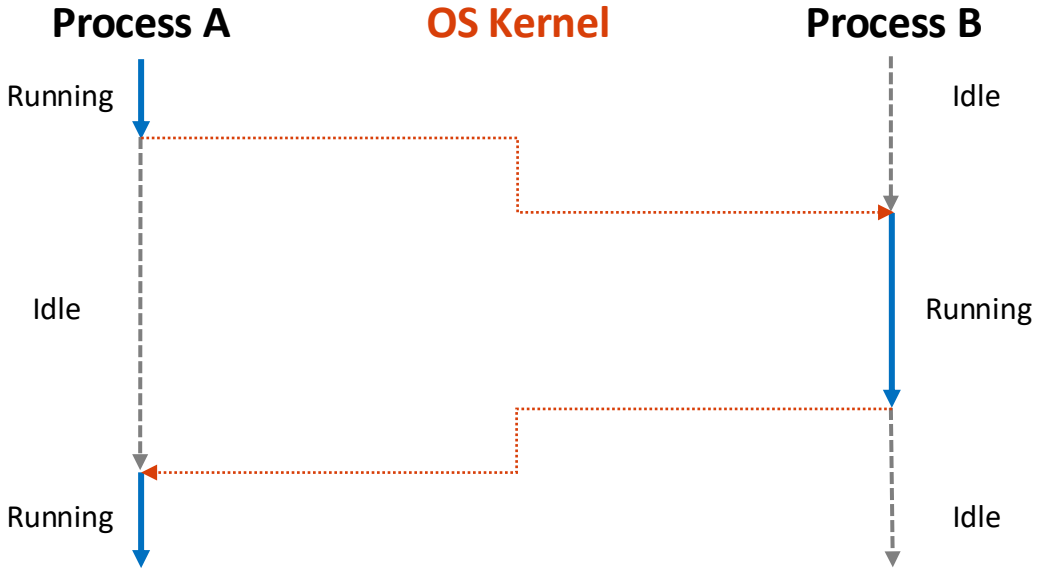
```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Deposit, info);
      case OP_WITHDRAW:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Withdraw, info);
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
}
```

RECAP: CONTEXT SWITCH

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another

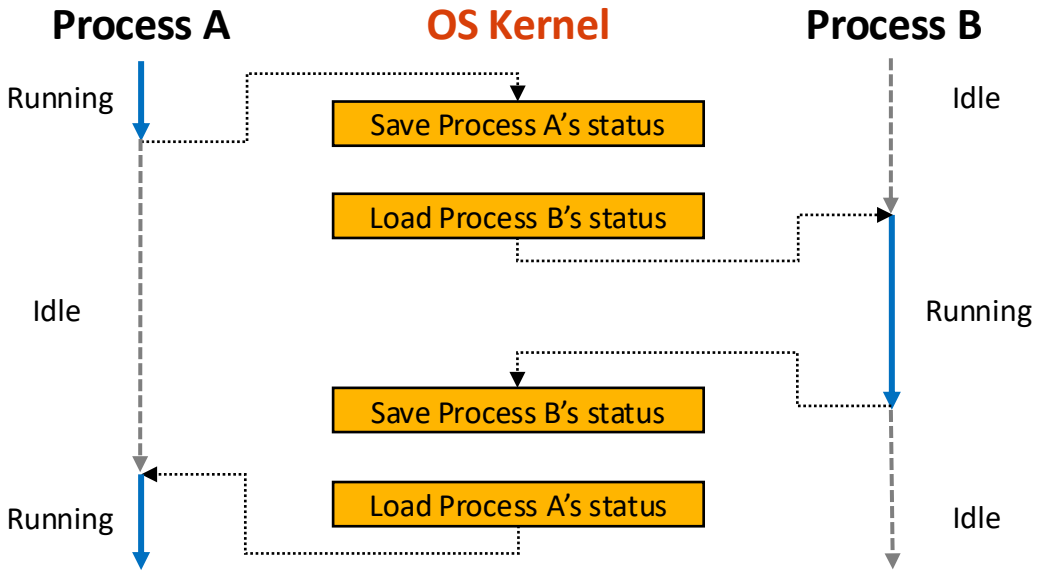
RECAP: CONTEXT SWITCH – CONT'D

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



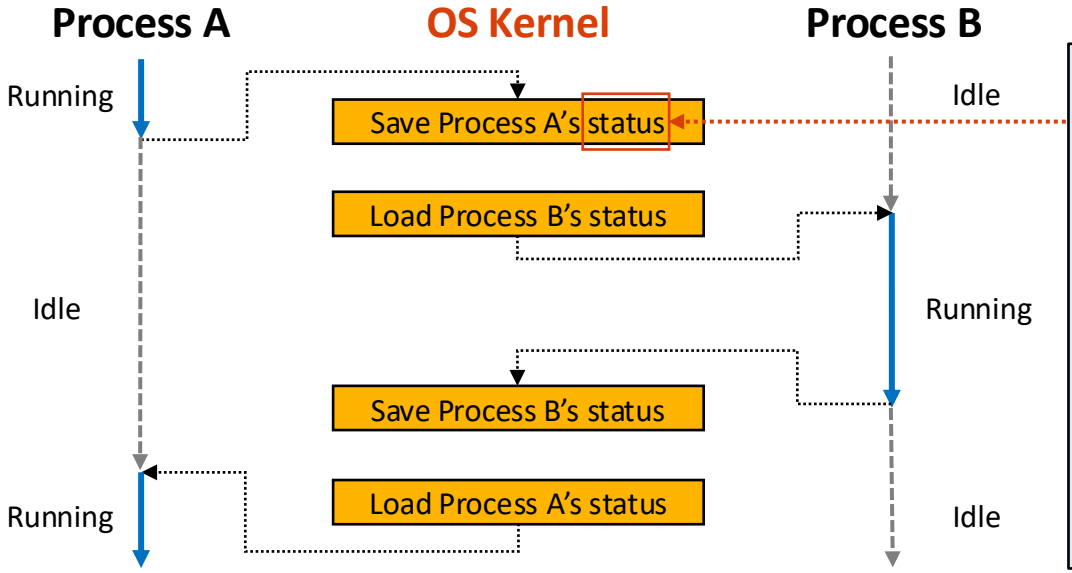
RECAP: CONTEXT SWITCH – CONT'D

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



RECAP: CONTEXT SWITCH – CONT'D

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



Recall: Process control block

A structure in OS that contains a set of information required to run a process on a CPU. Recall that Linux has *task_struct*.

- CPU#
- Program counter
- Instruction pointer
- Heap/stack pointer
- Process state [!]
- …

RECAP: OS SCHEDULER

- **(OS) Scheduler:**
  - **Definition:** An OS task (process) that manages the process scheduling activity

- **Implementation**

```
while ( <some condition,
        but eventually will be infinite>) {

   RunProcess( curProc );
   newProc = chooseNextProc();
   saveCurrentProc( curProc );
   LoadNextState( newProc );

}
```

  - It is also a process (an *infinite* loop)
  - The scheduler process terminates if we *stop* (turn-off) a computer

RECAP: OS SCHEDULER

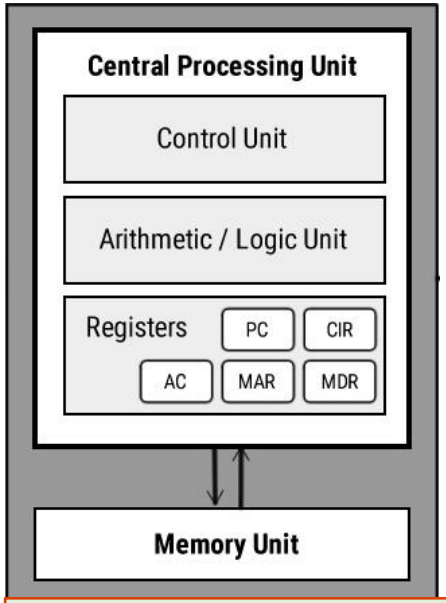
- **How OS scheduler works?**

```
while ( <some condition,
          but eventually will be infinite>) {

  RunProcess( curProc );
  newProc = chooseNextProc();
  saveCurrentProc( curProc );
  LoadNextState( newProc );

}
```

  - RunProcess(): a CPU executes the machine code of "curProc"

PCB$_{curProc}$

CPU#
Prog. counter
Heap/Stack



curProc (a = 5 + 8)

Program counter (PC)

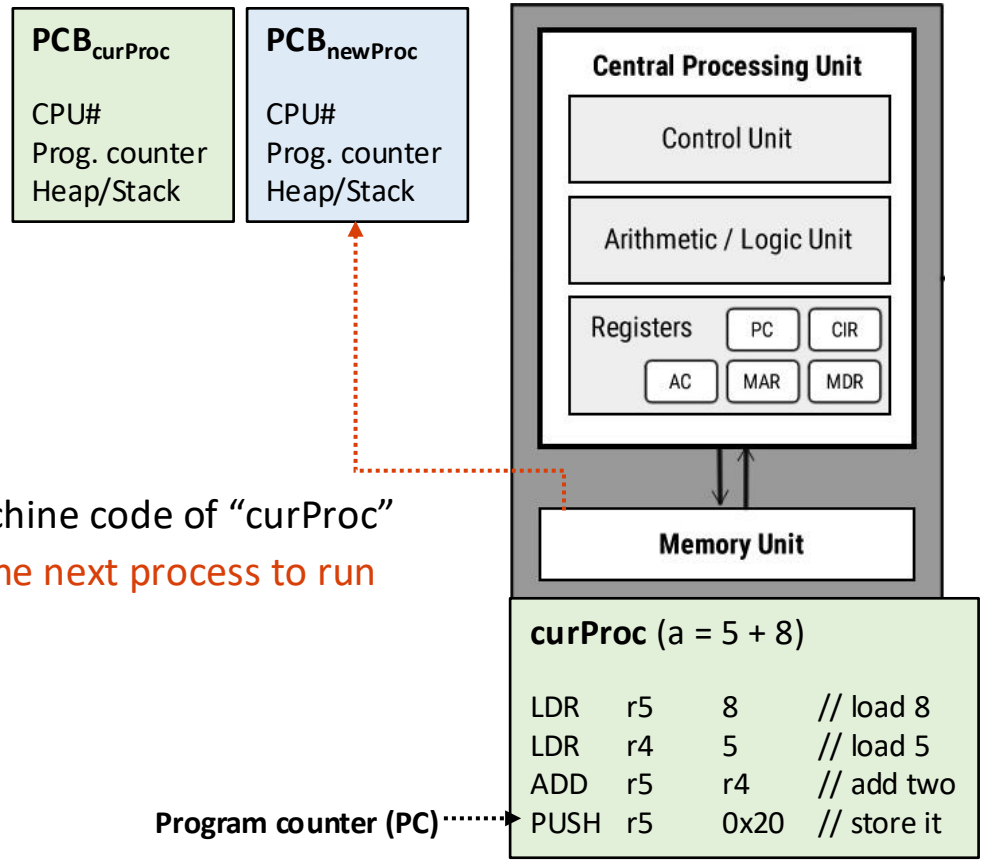
```
LDR   r5   8     // load 8
LDR   r4   5     // load 5
ADD   r5   r4    // add two
PUSH  r5   0x20  // store it
```

RECAP: OS SCHEDULER – CONT'D

- **How OS scheduler works?**

```
while ( <some condition,
          but eventually will be infinite>) {

  RunProcess( curProc );
  newProc = chooseNextProc();
  saveCurrentProc( curProc );
  LoadNextState( newProc );

}
```

  - RunProcess(): a CPU executes the machine code of "curProc"
  - chooseNextProc(): OS kernel selects the next process to run

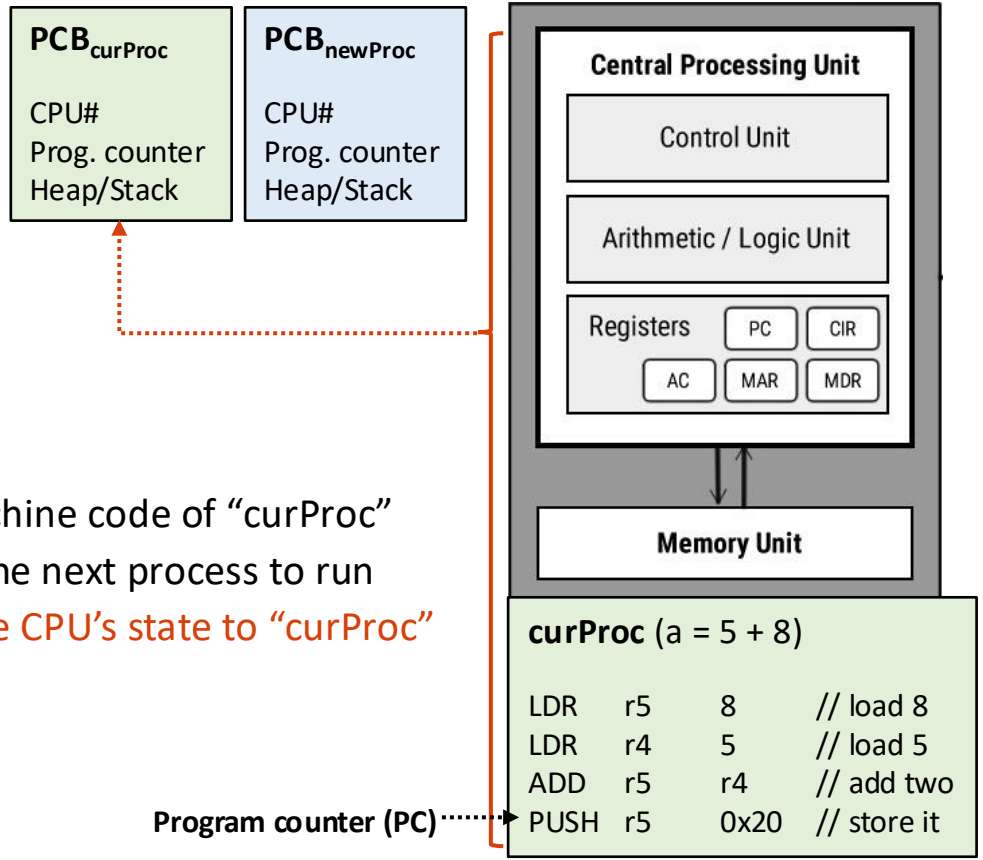


RECAP: OS SCHEDULER – CONT'D

- **How OS scheduler works?**

```
while ( <some condition,
          but eventually will be infinite>) {

  RunProcess( curProc );
  newProc = chooseNextProc();
  saveCurrentProc( curProc );
  LoadNextState( newProc );

}
```

  - RunProcess(): a CPU executes the machine code of "curProc"
  - chooseNextProc(): OS kernel selects the next process to run
  - saveCurrentProc(): OS kernel saves the CPU's state to "curProc"

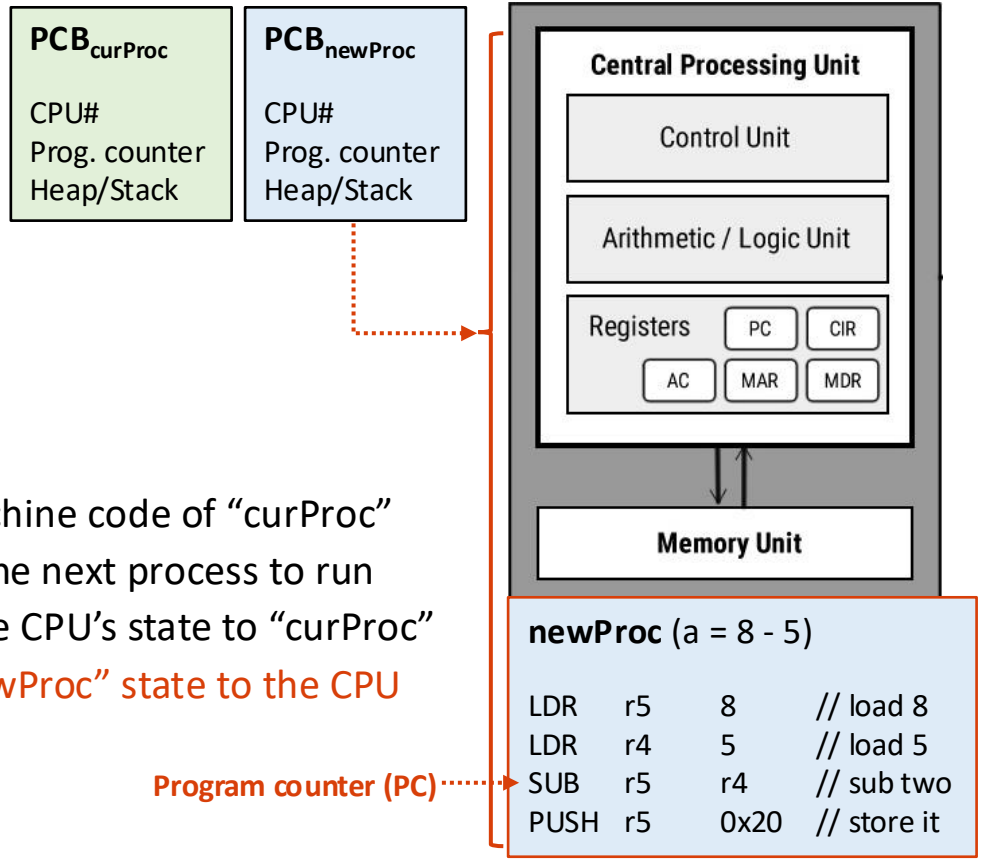


RECAP: OS SCHEDULER – CONT'D

- **How OS scheduler works?**

```
while ( <some condition,
          but eventually will be infinite>) {

  RunProcess( curProc );
  newProc = chooseNextProc();
  saveCurrentProc( curProc );
  LoadNextState( newProc );

}
```

  - RunProcess(): a CPU executes the machine code of "curProc"
  - chooseNextProc(): OS kernel selects the next process to run
  - saveCurrentProc(): OS kernel saves the CPU's state to "curProc"
  - loadNextState(): OS kernel stores "newProc" state to the CPU



RECAP: OS SCHEDULER – CONT'D

- **What triggers OS scheduling?**

```
while ( <some condition,
          but eventually will be infinite>) {

   RunProcess( curProc );
   newProc = chooseNextProc();   <····· Yield or interrupt triggers this code line
   saveCurrentProc( curProc );
   LoadNextState( newProc );

}
```

  - RunProcess(): a CPU executes the machine code of "curProc"
  - chooseNextProc(): OS kernel selects the next process to run
  - saveCurrentProc(): OS kernel saves the CPU's state to "curProc"
  - loadNextState(): OS kernel stores "newProc" state to the CPU

PROBLEM OF MULTI-THREADED BANK SERVER

- **OS controls the concurrency**
  - Can't control context switch
  - Can't control the execution order
  - Can't guarantee the balance

Thread A: Deposit $200

1. LOAD r1 = $400

3. ADD r1 $200 ($600)
4. STORE r1 ($600)

Thread B: Deposit $100

2. LOAD r1 = $400

5. ADD r1 $100 ($500)
6. STORE r1 ($500)

Final Balance: $500?!

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Deposit, info);
      case OP_WITHDRAW:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Withdraw, info);
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
}
```

PROBLEM IN MANAGING RESOURCES

- **Race condition**

RACE CONDITION

- **Race condition**
  - **Definition:** an undesirable scenario in which
    - 2+ threads access a shared resource concurrently, *and*
    - The result depends on the order of execution
  - **Why this is dangerous?**
    - Does not always produce incorrect results
    - Difficult to reproduce consistently
  - **(In the deposit example) Problem**
    - Balance += amount is not one instruction
      - It loads the balance from memory
      - It then adds the amount to the balance
      - It finally stores the sum to the balance
    - A *context switch* can happen between any time

RACE CONDITION

- **Other real-world problems**
  - Table reservation systems – two people reserve the same table at the same time
  - Software version control – two developers edit the same file and push at the same time
  - Database systems – two people purchases the same product at the same time
  - … (shared resource + access it + modify it)

TOPICS FOR TODAY

- Part IV – Synchronization
  - Manage resources
    - Problem of sharing resources
    - Race condition
  - Provide abstraction
    - Atomic operation
    - Mutual exclusion (mutex)
  - Offer standard interface
    - Mutex C libraries
    - Critical section

ATOMIC OPERATION

- **Problem**
  - Deposit() is *indivisible*
  - A ctx switch can happen in the middle
  - Make sure to execute it at once

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Deposit, info);
      case OP_WITHDRAW:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Withdraw, info);
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
}
```

ATOMIC OPERATION

- **Problem**
  - Deposit() is *indivisible*
  - A ctx switch can happen in the middle
  - Make sure to execute it at once
- **Atomic operation**
  - Code must be run w/o interrupt
  - Either completes fully or no execution

```
void ProcessRequest(op, accountId, amount) {
  switch (op) {
    case OP_DEPOSIT:
      pthread_t *newTh = <mem alloc>;
      pthread_create(newTh, Deposit, info);
    case OP_WITHDRAW:
      pthread_t *newTh = <mem alloc>;
      pthread_create(newTh, Withdraw, info);
  }
}

void Deposit(accountId, amount) {
  account = GetAccount(accountId);
  account->balance += amount;
  StoreAccount(account);
}

int main(void) {
  int op = -1;
  int accountId, amount = -1, -1;

  while (1) {
    ReceiveRequest(&op, &accountId, &amount);
    ProcessRequest(op, accountId, amount);
  }

  return 0;        // code only reaches here if the server terminates
}
```

ATOMIC OPERATION – TWO APPROACHES

- **Hardware-level atomic operations**
  - x86 instruction: lock add [balance], amount
  - This only works for a simple operation (e.g., with single variable)

- **Software-level atomic operations**
  - Use a mutex for a complex code blocks (next slides)
  - Mutex works for multiple lines of code

MUTUAL EXCLUSION (MUTEX)

- Mutex
  - **Definition:** a lock ensures that only one thread can run a critical section at a time
  - **Three operations**
    - Lock before running a critical section
    - Unlock after running the critical section
    - Wait while someone locked the code

MUTUAL EXCLUSION (MUTEX)

- Mutex
  - **Definition:** a lock ensures that only one thread can run a critical section at a time
  - **Three operations**
    - Lock before running a critical section
    - Unlock after running the critical section
    - Wait while someone locked the code

```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
  switch (op) {
    case OP_DEPOSIT:
      …
  }
}

void Deposit(accountId, amount) {
  pthread_mutex_lock(&foo_mutex);        // lock before the atomic op.
  account = GetAccount(accountId);
  account->balance += amount;
  StoreAccount(account);
  pthread_mutex_unlock(&foo_mutex);      // unlock after the atomic op.
}

int main(void) {
  int op = -1;
  int accountId, amount = -1, -1;
  pthread_mutex_init(&deposit_lock, NULL);

  while (1) {
    ReceiveRequest(&op, &accountId, &amount);
    ProcessRequest(op, accountId, amount);
  }

  return 0;        // code only reaches here if the server terminates
```

MUTUAL EXCLUSION (MUTEX)

- Critical section
  - **Definition:** the code region protected by mutex lock and unlock (only one thread can be inside this)
  - **Three required properties**
    - Mutual exclusion: only one thread can be inside the critical section at a time
    - Progress: if no thread holds the lock, a waiting thread must be able to enter
    - Bounded waiting: no thread should wait forever

```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
  switch (op) {
    case OP_DEPOSIT:
      ...
  }
}

void Deposit(accountId, amount) {
  pthread_mutex_lock(&foo_mutex);        // lock before the atomic op.
  account = GetAccount(accountId);
  account->balance += amount;
  StoreAccount(account);
  pthread_mutex_unlock(&foo_mutex);      // unlock after the atomic op.
}

int main(void) {
  int op = -1;
  int accountId, amount = -1, -1;
  pthread_mutex_init(&deposit_lock, NULL);

  while (1) {
    ReceiveRequest(&op, &accountId, &amount);
    ProcessRequest(op, accountId, amount);
  }

  return 0;        // code only reaches here if the server terminates
```

RECAP: PROCESS VS. THREADS STATE

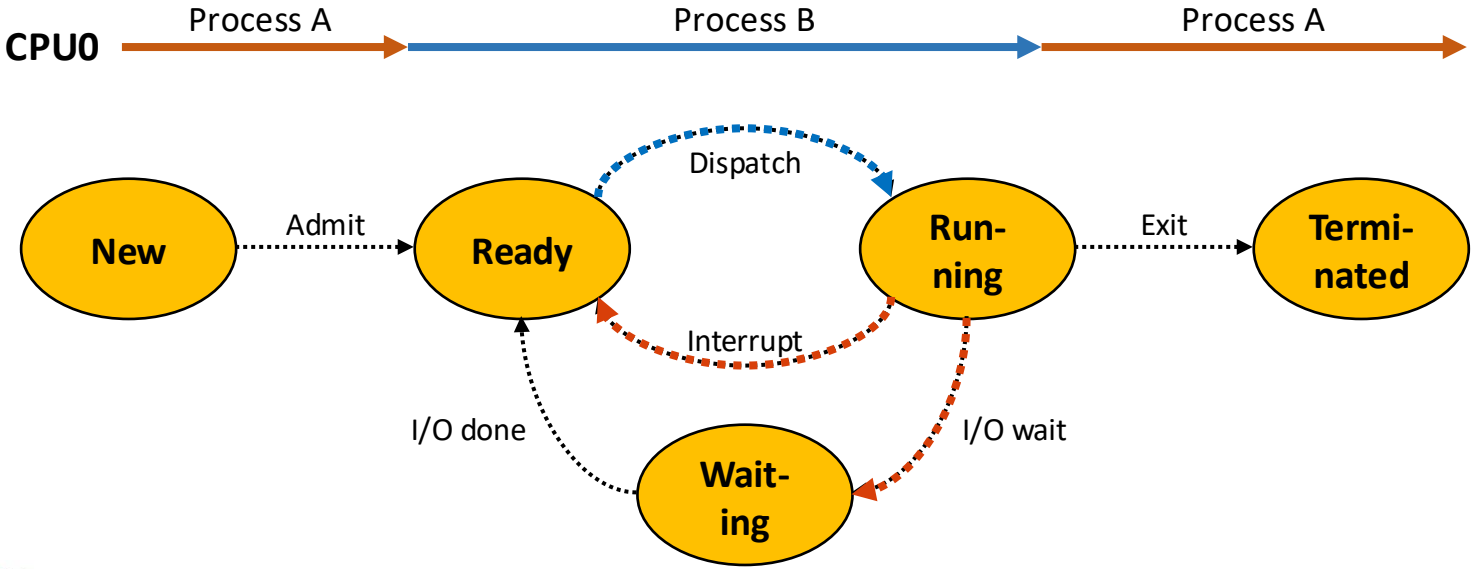
- A process can have five states:
  - **New:** a process (or thread) is being created (by fork())
  - **Ready:** the process is waiting to run
  - **Running:** the process is running on a CPU(or CPUs)
  - **Waiting:** the process is waiting for some events to occur (*e.g.*, a data loaded from storage)
  - **Terminated:** the process has finished execution; waiting for removal

RECAP: THREAD STATE

- A thread can have three states:
  - **Running:** the thread is running on a CPU(or CPUs)
  - **Waiting/Blocked:** the thread is waiting for events (*e.g.*, a data loaded from storage)
  - **Terminated:** the thread has finished execution

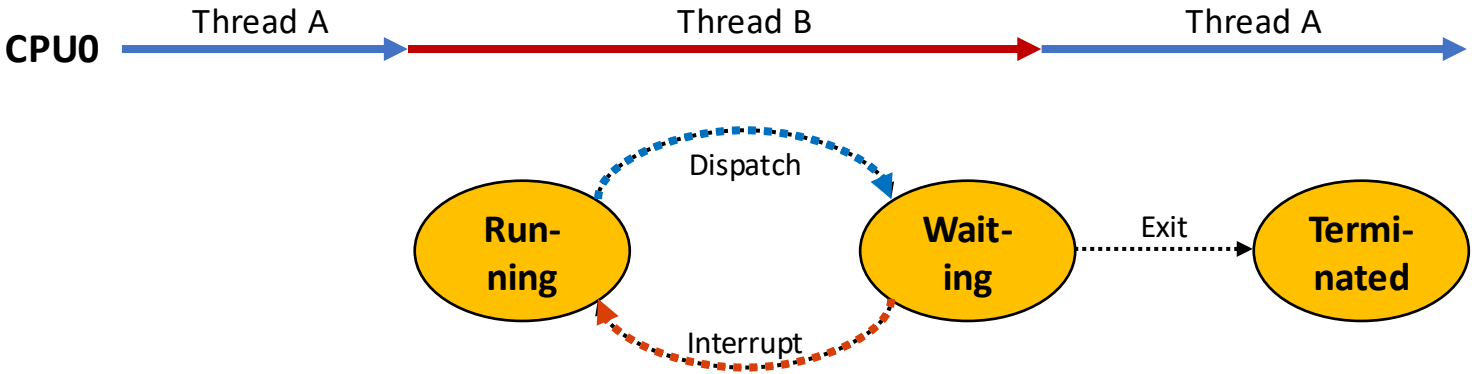
RECAP: PROCESS STATE TRANSITION

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



RECAP: THREAD STATE TRANSITION

- **Context switch**
  - **Definition:** OS stores the current thread's status and loads the new one
  - **Informal:** OS takes a CPU from one thread and gives it to another



MUTUAL EXCLUSION (MUTEX)

- Common mutex mistakes
  - **Mistake 1:** hold the lock too long
    - pthread_mutex_lock(&lock);
    - sleep(10000000);
    - pthread_mutex_unlock(&lock);
  - **Mistake 2:** forgot to unlock
    - pthread_mutex_lock(&lock);
    - if (error) return;
    - pthread_mutex_unlock(&lock);
  - **Mistake 3:** use the incorrect lock
    - pthread_mutex_lock(&lock_A);
    - account->balance += amount;
    - pthread_mutex_unlock(&lock_B);

TOPICS FOR TODAY

- Part IV – Synchronization
  - Manage resources
    - Problem of sharing resources
    - Race condition
  - Provide abstraction
    - Atomic operation
    - Mutual exclusion (mutex)
  - Offer standard interface
    - Mutex C libraries
    - Critical section

Thank You!

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu