PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Concurrency I

Baskin Engineering UC SANTA CRUZ

Notices

- **Lab 1** due 23:59 **Sunday January 19**
- **Assignment 1** due 23:59 **Sunday January 26**
- No Class Monday - Martin Luther King Jr. Day
- Change of Lecture Location

Baskin Engineering 101

From Wednesday January 22

Today's Lecture

- Processes & Threads Re-Cap
- Thread Models & Pools
- Shared Memory
- Introduction to Concurrency

- Introduction to Assignment 1

Process States & Transitions

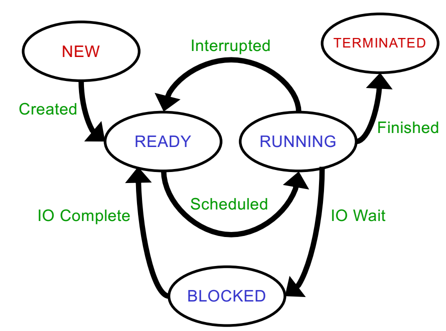
NEW:
The process (P) is being created

READY:
P is waiting to run on the CPU

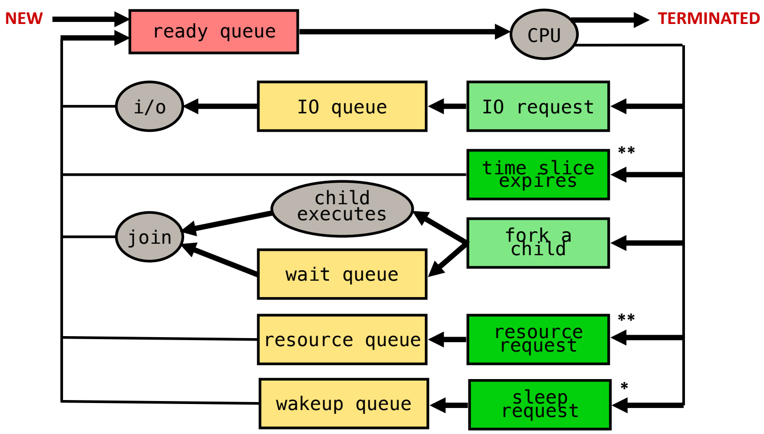
RUNNING:
P's instructions are being executed

BLOCKED:
P is waiting on IO to complete

TERMINATED:
P's execution is complete



Process Execution



* Lab 1 ** Lab 2

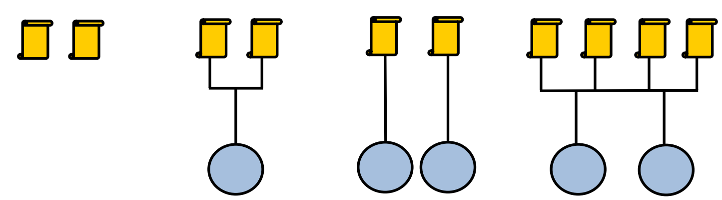
Recap: Processes & Threads

- **Processes**: “Heavyweight”
  - Expensive to create/switch/destroy
  - Owns resources (files, network sockets, etc.)
- **Kernel-level Threads**: “Lightweight”
  - At least one per process
  - If multiple, they share memory and resources
  - Has own stack, registers, program counter, id
  - Cheaper to create/switch/terminate than process
  - OS can schedule one to each logical core and can swap those that block
    - Super awesome feature, a justification for kernel-level threads all on it’s own ☺
- **User-level Threads**: “Inexpensive”
  - Implemented by user space libraries
  - Kernel is unaware of them
  - Very fast to create and manage
  - Blocked if the kernel thread (or process) running them is blocked
  - Modern libraries base user-level threads on kernel-level threads to benefit from multiprocessor machines

The Totally Tricky Thread Taxonomy

- **Kernel-level Thread**
  - Thread *class* managed by the operating system
  - Creation requires a system call unless created by the kernel itself
- **User-level Thread**
  - Thread *class* managed by a library external to operating system
  - Creation does not necessarily require a system call
  - Kernel is unaware of their existence
- **Kernel thread**
  - Thread *instance* created by the operating system kernel
  - Implicitly a Kernel-level Thread
- **User thread**
  - Thread *instance* created by a user process (code written / run by a user)
  - Can be a Kernel-level Thread or a User-level Thread

Thread Models (i.e. Mapping User-level Threads to Kernel-level Threads)



Many-to-None
Single process “Stand Alone” user-level threads - kernel is unaware of their existence

Many-to-One
Equivalent to single process user-level threads, and has the same advantages and shortcomings

One-to-One
Expensive, as one kernel-level thread is needed for each user-level thread

Many-to-Many
Best model - neither too slow, nor blocking, but difficult to program

User-level thread Kernel-level Thread

Thread Pools

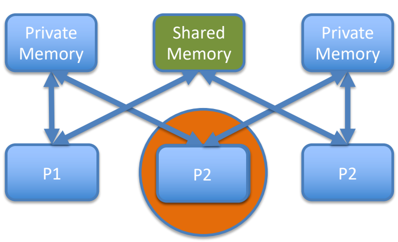
- If a (web or similar) server creates a new thread to serve each client:
  - Costs time to set up the new thread
  - Thread is thrown away afterwards
  - Wasteful of resources
  - Difficult to manage due to resource limits
- Instead, such a server can create a **pool of threads** to serve requests - threads are recycled back to the pool on completion
- This is easy to manage and implement, and only consumes a small amount of additional runtime resource
- Care must be taken to expunge state when the threads are recycled
  - **Why?**
    - Security

Inter Process Communication (IPC)

- Consider two processes on a Single CPU system
- How do they communicate with each other?
  - i.e. how do they share data?

Shared Memory

- Consider two processes on a Single CPU system



Unix / Linux Shared Memory System Calls

- `ftok` Generate a unique key
- `shmget` Get an identifier for a shared memory segment
- `shmat` Attach to the shared memory segment
- `shmdt` Detach from the shared memory segment
- `shmctl` Control (e.g. delete) the shared memory segment

http://man7.org/linux/man-pages/man2/shmget.2.html

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>

int main()
{
  // ftok to generate unique key key_t
  key = ftok("shmseg",65);

  // shmget returns an identifier in shmid
  int shmid = shmget(key,1024,0666|IPC_CREAT);

  // shmat to attach to shared memory
  char *str = (char*) shmat(shmid,(void*)0,0);

  // copy some data into shared memory
  strcat(str, "Hello CSE130!");
  printf("Written to shared memory\n");

  //detach from shared memory
  shmdt(str);

  return 0;
}
```

```
$ ./producer
Written to shared memory
```

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int main()
{
  // ftok to generate unique key
  key_t key = ftok("shmseg",65);

  // shmget returns an identifier in shmid
  int shmid = shmget(key,1024,0666|IPC_CREAT);

  // shmat to attach to shared memory
  char *str = (char*) shmat(shmid,(void*)0,0)
  printf("Data read: %s\n",str);

  //detach from shared memory
  shmdt(str);

  // destroy the shared memory
  shmctl(shmid,IPC_RMID,NULL);

  return 0;
}
```

```
$ ./consumer
Data read: Hello CSE130!
```



```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>

int main()
{
  // ftok to generate unique key key_t
  key = ftok("shmseg",65);

  // shmget returns an identifier in shmid
  int shmid = shmget(key,1024,0666|IPC_CREAT);

  // shmat to attach to shared memory
  char *str = (char*) shmat(shmid,(void*)0,0);

  // copy some data into shared memory
  strcat(str, "Hello CSE130!");
  printf("Written to shared memory\n");

  //detach from shared memory
  shmdt(str);

  return 0;
}
```

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
```

```
$ ./producer
Written to shared memory
$ ./producer
Written to shared memory
$ ./consumer
Data read: Hello CSE130!Hello CSE130!
```

```c
  // shmat to attach to shared memory
  char *str = (char*) shmat(shmid,(void*)0,0)
  printf("Data read: %s\n",str);

  //detach from shared memory
  shmdt(str);

  // destroy the shared memory
  shmctl(shmid,IPC_RMID,NULL);

  return 0;
}
```


POSIX Shared memory System Calls

- `shm_open` Create and open a new shared memory object or open an existing object
- `ftruncate` Set the size of the shared memory object (a newly created shared memory object has a length of zero)
- `mmap` Map the shared memory object into the virtual address space of the calling process
- `munmap` Unmap the shared memory object from the virtual address space of the calling process
- `shm_unlink` Remove a shared memory object by name
- `close` Close the file descriptor allocated by `shm_open` when it is no longer needed

http://man7.org/linux/man-pages/man7/shm_overview.7.html

Concurrency

Concurrency in Computer Systems

- ***"The actual or apparent simultaneous execution of threads"***
  - Actual = multicore and/or multiprocessors
  - Apparent = time slicing
  - Today's Reality = both
- Threads running concurrently in the same process typically want to access shared data
- Unfortunately, sharing data in an uncontrolled fashion allows that shared data to become:
  - Inconsistent
  - Incorrect
  - Invalid
  - Etc. etc. ☹ ☹ ☹

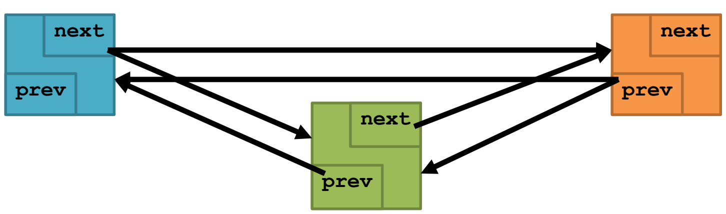
Example I

Consider this execution sequence:

Thread #1	Thread #2
varriable++;	**varriable--;**
LDA #0x0111	LDA #0x0111
INCA	DECA
STA #0x0111	STA #0x0111

Example II

Inserting into a doubly-linked list



List is inconsistent!

If reading as a handout, no this list in not inconsistent.
You'll have to watch the class recording to see the animation.

Critical Sections

- Sections of the program (short sequences of ***machine instructions***) that should only be executed by one thread at a time
- Consider a **stack** shared by many threads:
  - All threads should see a consistent state
  - `pop()` and `push()` operations are critical sections
- ***We (the OS) must control execution in <u>all</u> critical sections***

The Critical Section Problem

- We need a formalism that allows threads to arbitrate access to critical sections:
  - **Entry section** (used to negotiate entry to the critical section)
  - **Critical section** (do critical task)
  - **Exit section** (signal critical task completed)

Critical Section Requirements

- **Mutual Exclusion**:
  - Only one thread in critical section at a time
- **Progress**:
  - If a thread wishes to enter an idle critical section, then only threads in entry or exit section may help decide if it is allowed to do so
    - e.g. if a thread doesn't want access to a CS at the moment, it shouldn't stop other threads from accessing it
- **Bounded Wait**:
  - There exists a finite bound on the number of times a waiting thread must *"stand-aside"*

Software Solutions

- Hmmm, tricky…
- Even `variable++` does not translate into an atomic operation
- Typically a **busy-wait** solution is used

Two Thread Solution I

- Consider a solution involving only two threads (x & y)
- A first approach is to alternate each thread: x, y, x, y, x, y, …
- **Is this ok?**
- Which requirements are not met?
- No **progress** made if thread x wants to enter the critical section twice and y does not want to enter at all

Shared variables:

```
int turn;
```

Thread x:

```
while (turn != x) {
  yield();
}
// Critical Section
...
turn = y;
// Non-critical
...
```

Two Thread Solution II

- Modify by adding more state
- Record which threads want to enter their critical section
- Is this ok?
- **NO!** both threads can set their flags to true *"at the same time"* as the assignment is not protected ☹

Deadlock!

Shared variables:

```
bool flags[<some size>];
```

Thread x:

```
flags[x] = true;
while (flags[y]) {
  yield();
}
// Critical Section
...
flags[x] = false;
// Non-critical
...
```


Two Thread Correct Solution

- If solutions I and II are combined, then we can satisfy all three requirements
- Use the boolean array from solution II to record which threads (x & y) wish to enter their critical section
- Resolve deadlock with the alternating approach from solution I

Peterson's Two Thread Solution

Shared variables:

```
bool flags[<some size>];
int turn;
```

Thread x:

```
flags[x] = true;
turn = y;
while (flags[y] && turn == y) {
  yield();
}
// Critical Section
...
flags[x] = false;
// Non-critical
...
```

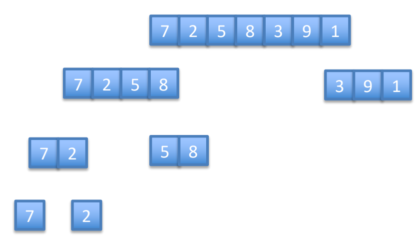
Thread y:

```
flags[y] = true;
turn = x;
while (flags[x] && turn == x) {
  yield();
}
// Critical Section
...
flags[y] = false;
// Non-critical
...
```


https://en.wikipedia.org/wiki/Peterson%27s_algorithm

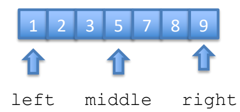
Assignment 1

- Merge Sort



https://opendsa-server.cs.vt.edu/embed/mergesortAV

Assignment 1




```
merge(array, left, middle, right) { /* magic! */ }

mergeSort(array, left, right) {
  if (left < right) {
    middle = (right-left)/2
    mergeSort(array, left, middle)
    mergeSort(array, middle+1, right)
    merge(array, left, middle, right)
  }
}
```


Next Lecture

- Introduction to Concurrency Cont.
- Semaphores