PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Processes & Threads II

Baskin Engineering UC SANTA CRUZ

Notices

- **Administration 1 & 2** due 23:59 **Wednesday January 15**
- **Lab 1** due 23:59 **Sunday January 19 CHANGED!**
- **Assignment 1** available today, due 23:59 **Sunday January 26**

Today's Lecture

- Brief Processes Re-Cap
- Introduction to Threads
- More Lab 1 Secret Sauce 

Process States & Transitions

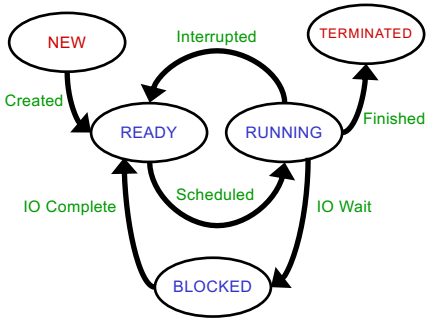
NEW:
The process (P) is being created

READY:
P is waiting to run on the CPU

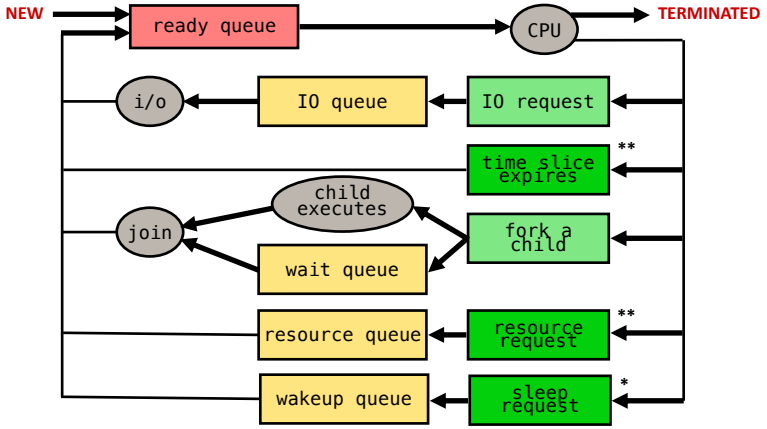
RUNNING:
P's instructions are being executed

BLOCKED:
P is waiting on IO to complete

TERMINATED:
P's execution is complete

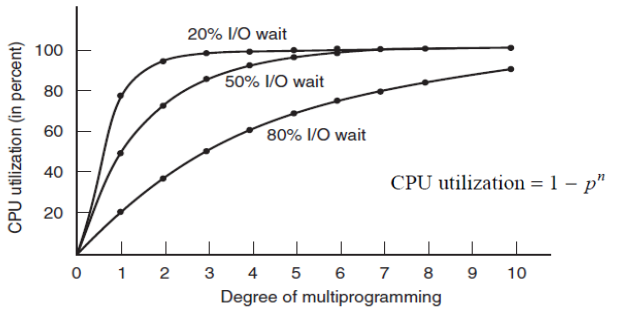


Process Execution



* Lab 1 ** Lab 2

Modeling Multiprogramming



p = % time spent waiting for I/O
n = no. of processes

Probability that all **n** processes are waiting for I/O = p^n

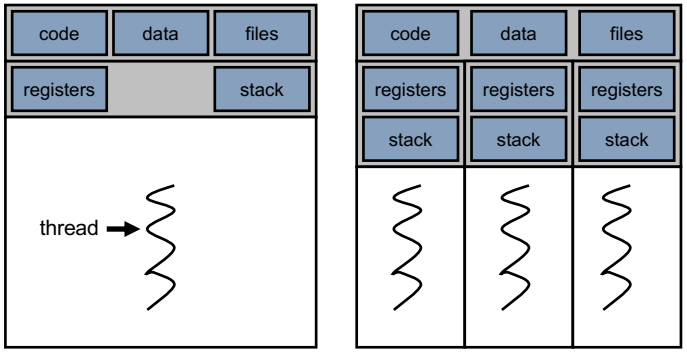
Switching Processes is Expensive

- We want to avoid the expensive ***context switch***
- Also, inter processes communication (IPC) is slow and expensive, mainly because of the context switch
- It would be simpler to have shared memory - but we often can't and usually shouldn't do that with processes for security reasons
- One solution is…

Threads

- We'd like to have more than one sequence of execution - **"light-weight-process"** (LWP) - per process
- One thread might monitor user input, a second update the screen, a third write backup files to the disk
- Coordination and communication are easier as threads share the process context; that is, the same memory, files, peripherals, etc.
- e.g. A Web Browser:
  - A separate thread for each HTTP request
  - Thread(s) to render the page(s)
  - Thread(s) for user input
- Processes become 'containers' for related threads
  - Threads share access to common memory and data structures
  - However, threads must be managed to make sure they don't corrupt the shared data
- Each process remains isolated from other processes and their address spaces

Threads



single-threaded process

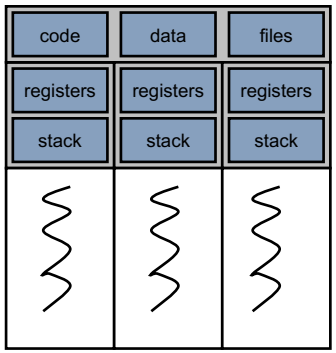
multi-threaded process

Benefits

- **Responsiveness**: If one thread is blocked, another can still execute without slow and costly context switch (OS dependent)
- **Sharing**: Coordination easier as they share the same memory, files, etc. so no expensive IPC
- **Efficiency**: Creating and switching threads is cheaper in time and resources than creating and switching between multiple processes
- **Utilization**: On machines with multiple processors and shared memory (between the processors), we can schedule a thread on each processor from the same process
- **Modularity**: Writing smaller code units to do specific tasks is always better design

Thread Issues

- Many threads can execute in one process ☺
- Threads share process state (except for register values and stack)
- => they can interfere with each other's execution ☹



multi-threaded process

Implementing Threads

- Early time sharing Operating Systems did not support threads, they only scheduled processes
- Implementation Options?
  - As user level libraries
    - **"user-level threads"** => cheap to implement
  - Redesign the OS to deal with threads as first class entities
    - **"kernel-level threads"** => expensive to implement

User-level Threads

- Implemented by thread libraries above/outside the OS kernel
- Thread creation, scheduling, switching etc. is performed at user-level, no special privileges required, no OS involvement
- Very fast, as there are no system calls, so no context switches
  - Remember: A system call switches control to OS
- **CAVEAT**: If one thread blocks on a system call, then the entire process will be blocked

Kernel-level Threads

- Kernel knows about threads:
  - Creates, schedules, switches etc.
- If one kernel-level thread blocks, the kernel will schedule another from the same process - assuming we are still within the process' time slice (This is CPU scheduling, covered in later lectures and Lab 2)
- Can take advantage of multiple processors
- **Thread Control Block** (TCB) exists to track thread id (tid), counter, stack, registry data, etc. etc.
- CAVEAT: Slower than user-level threads, but faster than a process context switch

POSIX Threads

- Portable Operating System Interface (POSIX)
- International standard for OS compatibility
  - IEEE Std 1003.1c-1995
- POSIX Threads
  - Known as "**pthreads**"
- ~100 functions, all prefixed `pthread_`
  - Thread management - creating, joining, waiting, etc.
  - Semaphores, Mutexes, Condition Variables (covered in later lectures and Lab 2)
  - Synchronization between threads using read/write locks and barriers
- Whatever OS your are on, the pthreads *interface* is identical ☺
  - Though the *implementation* may be different


```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX_CHILDREN      10
#define MAX_SLEEP_SECONDS 32

void forkChildProcess() {
    switch(fork()) {
    case -1:
        fprintf(stderr, "ERROR: Fork failed!\n");
        exit(EXIT_FAILURE);
    case 0:
        srand(getpid());
        int seconds = rand() % MAX_SLEEP_SECONDS;
        printf("Child PID %d, sleeping for %2d seconds\n",
            getpid(), seconds);
        sleep(seconds);
        printf("Child PID %d, finished after %2d seconds\n",
            getpid(), seconds);
        exit(EXIT_SUCCESS);
    }
}

int main(int argc, char* argv[]) {
    int status = 0;

    printf("Parent Running PID %d\n", getpid());
    for (int i = 0 ; i < MAX_CHILDREN; i++) {
        forkChildProcess();
    }

    while (wait(&status) > 0);
    return(EXIT_SUCCESS);
}
```

"Multi-Process"

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

#define MAX_THREADS       10
#define MAX_SLEEP_SECONDS 32

void *threadInit(void *param){
    long tid = (long) param;
    srand(tid);
    int seconds = rand() % MAX_SLEEP_SECONDS;
    printf("Thread %ld, sleeping for %2d seconds\n",
        tid, seconds);
    sleep(seconds);
    printf("Thread %ld, finished after %2d seconds\n",
        tid, seconds);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[MAX_THREADS];

    printf("Parent Running PID %d\n", getpid());
    for (long i = 0; i < MAX_THREADS; i++) {
        int rc = pthread_create(
            &threads[i], NULL, threadInit, (void *) i);
        if (rc) {
            fprintf(stderr,
                "ERROR: pthread_create() returned %d\n", rc);
            exit(EXIT_FAILURE);
        }
    }
    pthread_exit(NULL);
    return(EXIT_SUCCESS);
}
```

"Multi-Threaded"

```
Parent Running PID 31210

Child PID 31211, sleeping for 26 seconds
Child PID 31212, sleeping for  6 seconds
Child PID 31213, sleeping for 29 seconds
Child PID 31214, sleeping for 10 seconds
Child PID 31215, sleeping for 19 seconds
Child PID 31216, sleeping for 29 seconds
Child PID 31217, sleeping for 15 seconds
Child PID 31218, sleeping for 18 seconds
Child PID 31219, sleeping for 10 seconds
Child PID 31220, sleeping for  8 seconds

Child PID 31212, finished after  6 seconds
Child PID 31220, finished after  8 seconds
Child PID 31214, finished after 10 seconds
Child PID 31219, finished after 10 seconds
Child PID 31217, finished after 15 seconds
Child PID 31218, finished after 18 seconds
Child PID 31215, finished after 19 seconds
Child PID 31211, finished after 26 seconds
Child PID 31213, finished after 29 seconds
Child PID 31216, finished after 29 seconds
```

```
Parent Running PID 31572

Thread 0, sleeping for  7 seconds
Thread 1, sleeping for  7 seconds
Thread 3, sleeping for 26 seconds
Thread 7, sleeping for 21 seconds
Thread 4, sleeping for 29 seconds
Thread 8, sleeping for 24 seconds
Thread 9, sleeping for  3 seconds
Thread 5, sleeping for 18 seconds
Thread 6, sleeping for 17 seconds
Thread 2, sleeping for  5 seconds

Thread 9, finished after  3 seconds
Thread 2, finished after  5 seconds
Thread 0, finished after  7 seconds
Thread 1, finished after  7 seconds
Thread 6, finished after 17 seconds
Thread 5, finished after 18 seconds
Thread 7, finished after 21 seconds
Thread 8, finished after 24 seconds
Thread 3, finished after 26 seconds
Thread 4, finished after 29 seconds
```

```
PTHREAD_EXIT(3)          Linux Programmer's Manual          PTHREAD_EXIT(3)

NAME
       pthread_exit - terminate calling thread

SYNOPSIS
       #include <pthread.h>

       void pthread_exit(void *retval);

       Compile and link with -pthread.

DESCRIPTION
       The pthread_exit() function terminates the calling thread and returns
       a value via retval that (if the thread is joinable) is available to
       another thread in the same process that calls pthread_join(3).

       Any clean-up handlers established by pthread_cleanup_push(3) that
       have not yet been popped, are popped (in the reverse of the order in
       which they were pushed) and executed.  If the thread has any thread-
       specific data, then, after the clean-up handlers have been executed,
       the corresponding destructor functions are called, in an unspecified
       order.

       When a thread terminates, process-shared resources (e.g., mutexes,
       condition variables, semaphores, and file descriptors) are not
       released, and functions registered using atexit(3) are not called.

       After the last thread in a process terminates, the process terminates
       as by calling exit(3) with an exit status of zero; thus, process-
       shared resources are released and functions registered using
       atexit(3) are called.

RETURN VALUE
       This function does not return to the caller.

ERRORS
       This function always succeeds.
```

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 multi-processor / multi-core hardware**

Lab 1 Secret Sauce

- In the `timer_sleep()` function:
  - Put the current thread to sleep and immediately return
- At regular points in the future:
  - Wake up sleeping threads at or past their wakeup time

Lab 1 Saucier Secrets (but still not a complete solution)

- In `timer.c`
  - Define a list for sleeping threads
  - Initialise this list somewhere appropriate
- In `timer_sleep()`
  - Store the wakeup time in the current thread
  - Put the current thread into the list of sleeping threads
  - Put the current thread to sleep
- In the **timer interrupt handler** (you still need to find where this is)
  - Iterate over the list of sleeping threads
  - If any are at or past their wakeup time…
    - Wake them up
    - Remove them from the sleeping threads list

Lab 1 Saucier Secrets (alternate solution)

- In `timer_sleep()`
  - Store the wakeup time in the current thread
  - Put the current thread to sleep
- In the **timer interrupt handler** (you still need to find where this is)
  - Iterate over ALL threads
  - Find the sleeping threads
  - If any are at or past their wakeup time…
    - Wake them up

Next Lecture

- More on Threads
- Shared Memory

- Introduction to Assignment 1