Sistemas Operacionais

Trabalho prático I

Biblioteca de Threads CThreads 16.2

Professor Alexandre Carissimi

Caroline Knorr Carvalho – 00229753

Guilherme de Oliveira Tassinari – 00231060

Universidade Federal do Rio Grande do Sul

- Ccreate

```
int ccreate (void* (*start)(void*), void *arg){
 if(firstTime == 0){
    firstTime = 1;
    initMainThread();
  }
  if(start == NULL) return ERROR;
  ucontext_t * threadContext = malloc(sizeof(ucontext_t)); //Aloca memória para um
contexto
  createContext(threadContext, start); //Cria o contexto para a nova thread


  TCB_t * newThread = malloc(sizeof(TCB_t));
  createThread(newThread, threadContext);
  readyThread(newThread);
  return newThread->tid;
}
```

Em funcionamento : SIM

Descrição do código: primeiramente, através de uma variável global, é testado se é a primeira vez que a biblioteca está sendo utilizada. Se o for, o TCB da thread main é criado. Logo após é criado o contexto, seguido da thread que, por fim, é adicionada à fila de aptos.

- Cyield

```
int cyield(void){
  int * isCommingBack = malloc(sizeof(int));
  *isCommingBack = 1;
  if(yield() == SUCCESS){ //Libera a execução e passa a thread para o estado apto
    if(*isCommingBack == 1){
      *isCommingBack = 0;
      sortAndExecuteThread(); //Executa uma nova thread
    }
  }
  free(isCommingBack);
  return SUCCESS;
}
```

Em funcionamento: SIM

Descrição do código: Logo no ínicio é alocada uma memória que guarda um valor inteiro representando se a execução corrente está sendo feita antes ou após a cedência voluntária. Caso a cedência ainda não tenha sido feita ela é realizada, o valor da variável de controle é alterado e uma nova thread é sorteada para ser executada. Como a troca de contexto é feita dentrod o método yield, quando ocorrer o retorno da execução, a variável de controle impedirá que que uma nova thread seja sorteada e a execução ocorrerá normalmente.

- Csem_init

```
int csem_init (csem_t *sem, int count){
  if(sem == NULL){
    sem = malloc(sizeof(csem_t*));
  }
  sem->count = count;
  sem->fila = malloc(sizeof(PFILA2));
  CreateFila2(sem->fila);
  return SUCCESS;
}
```

Em funcionamento: SIM

Descrição do código: Verifica-se se o semáforo já foi inicializado. Se não o foi, um espaço de memória é alocado. O contador é inicializado com o valor passado e a fila de espera é criada.

- Csignal

```
int csignal(csem_t *sem){
  sem->count = sem->count + 1; //Incrementa o contador do semaforo
  if(sem->count < 1){ //Se ainda tiver alguém esperando na fila, desbloqueia.
    FirstFila2(sem->fila);
    TCB_t * thread = (TCB_t*)GetAtIteratorFila2(sem->fila); //Pega uma thread da
fila do semaforo
    unblockThread(thread); //Coloca a thread na fila de aptos
    DeleteAtIteratorFila2(sem->fila); //Deleta a thread da fila do semaforo
  }
  return SUCCESS;
}
```

Em funcionamento: SIM

Descrição do código: O contador do semáforo é incrementado. Após, caso ele ainda tenha alguém esperando sua liberação, é retirada uma thread da fila de espera e de bloqueados e é colocada na fila de aptos.

- Cwait

```
int cwait(csem_t *sem){
  sem->count = sem->count - 1;
  if(sem->count < 0){ //Se o recurso estiver alocado
    int * isCommingBack = malloc(sizeof(int));
    *isCommingBack = 1;
    TCB_t * thread = blockThread(); //Bloqueia a thread
    AppendFila2(sem->fila, thread); //Coloca a thread na fila do semáforo
    getcontext(&(thread->context)); //salva o contexto
    if(*isCommingBack == 1){
      *isCommingBack = 0;
      sortAndExecuteThread(); //Executa uma nova thread
    }
    //Quando esta thread for executada novamente, isCommingBack será zero, e ela
continuará sua execução normalmente
    free(isCommingBack);
  }
  return SUCCESS;
}
```

Em funcionamento: SIM

Descrição do código: O contador do semáforo é decrementado. Caso o recurso estivesse alocado, a thread atual é colocada na fila do semáforo e na fila de bloqueados. Por fim seu contexto é salvo e outra thread é sorteada para execução.

- Cjoin

```
int cjoin(int tid){
  return waitForThread(tid);
}

 int waitForThread(int tid){
   initFila(&joints);
   joint * join;
   FirstFila2(joints);
   do {  //Varre a fila de espera procurando por alguma thread esteja esperando
 a thread solicitada
     join = (joint*)GetAtIteratorFila2(joints);
     if(join == NULL) break;
     if(join->threadToWait == tid){ //Se ja houver alguem na espera, retorna
 erro, pois não podem haver duas threads esperando a mesma thread
       return ERROR;
       break;
     }
   }while(NextFila2(joints) == SUCCESS);
   //Caso não hajam problemas, bloqueia a thread atual e a coloca na fila de
 espera da thread solicitada
   TCB_t * currentThread = blockThread();
   join = malloc(sizeof(joint));
   join->thread = currentThread->tid;
   join->threadToWait = tid;
   AppendFila2(joints, join); //Registro na fila de espera
   int * isCommingBack = malloc(sizeof(int));
   *isCommingBack = 1;
   getcontext(&(currentThread->context)); //A execução continuará daqui quando a
 thread for liberada
   if(*isCommingBack == 1){
     *isCommingBack = 0;
     sortAndExecuteThread(); //Sorteia a proxima thread a ser executada
   }
   free(isCommingBack);
   return SUCCESS;
 }
```

Em funcionamento: SIM

Descrição do código: A fila de Joints (estruturas que guardam a informação de threads que esperam outras threads) é inicializada. Depois verifica-se se não existe nenhuma outra thread já esperando pela thread solicitada. Se houver, é retornado

um código de erro. Caso contrário, a thread em execução atualmente é colocada na fila de bloqueadas e uma nova joint é inserida na fila de joints. Por fim, uma nova thread é sorteada e executada.

- Cidentify

```
int cidentify(char *name, int size){
  char fullNames[] = "Guilherme Tassinari(231060) & Caroline Knorr(229753)";
  char reducedNames[] = "GOT(231060) & CKC(229753)";
  if(size < sizeof(fullNames)/sizeof(char)){
    if(size < sizeof(reducedNames)/sizeof(char)){
      return ERROR; //Retorna erro se o tamanho passado for muito pequeno
    } else {
      name = reducedNames; //escreve nomes abreviados se o tamanho permitir
    }
  } else {
    name = fullNames; //Escreve os nomes completos se o tamanho permitir
  }
  return SUCCESS;
}
```

Em funcionamento: SIM

Descrição do código: Duas possíveis strings são criadas. Uma completa e outra reduzida. É testado se o espaço passado é suficiente para a escrita dos nomes. Se alguma string encaixar no espaço, ela é passada. Se não, um erro é retornado.

- Testes
  - Create: No teste create são criadas 5 funções distintas que são passadas para 4 threads distintas. As threads, por sua vez, são executadas sequencialmente através de chamdas cjoin() sequenciais na main. Uma das threads cria, dentro de si mesma, uma nova thread, e executa uma chamada cjoin() para esperar sua execução. Espera-se que as threads sejam executadas corretamente de acordo com a ordem da main e que a thread que chama o seu cjoin() internamente só termine após o término de sua thread filha.
  - Semaphores: Este teste tem por objetivo testar o funcionamento dos semáforos. São definidas 5 funções distintas. Uma delas possui chamadas cwait() e csignal() internas e comandos printf() avisando quando o recurso é solicitado, alocado e liberado. São criadas múltiplas threads

para cada uma das funções, em especial quantidade a função com chamadas de semáforos. Espera-se que não hajam mensagens de alocação de recursos sequenciais, mas sim que as mesmas sejam intercaladas com mensagens de liberação de recursos, simbolizando que apenas uma thread por vez pode utilizar o mesmo.

- Dificuldades:
  - ucontext_t : a utilização correta da biblioteca de contextos apresentou certo desafio, principalmente em trocas e resumos de contexto, apesar de ter sido superado em pouco tempo.
  - Biblioteca support: apresentou dificuldades por débito técnico da dupla. Má utilização e interpretação errônea da documentação causaram problemas.